TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 3

Neural Network Basics

3.0 Introduction

It is often tempting to treat NNs as "blackboxes" without really understanding deeply why they work, what their limitations are and when/how they can fail. In our experience, the "blackbox" approach is not productive because it leaves you helpless when things go wrong – which is often when tackling real-world problems!

Instead, our goals in this and subsequent lectures is to give you:

- **An intuition** for how NNs behave. This is done in the lecture notes, Quizzes (you should complete as many as you can) and Labs (you should complete every single lab). Our hope is that you end up with a deep mental model of NNs, so that you become surprised if something does/doesn't work. This feeling of "surprise" is what helps you decide what to do next when something goes wrong.

- **A set of successful recipes** that we have found to work in practice for time-series forecasting. Our hope is that you can build on our successes and avoid our mistakes.

3.1 The Perceptron

Artificial Neural Networks (NNs) were originally proposed in the 1940's as abstract mathematical models of biological neurons, the cells in brains of animals. Since then, NNs have left their original biological inspiration, and developed independently as a tool for solving a wide variety of pattern recognition and learning problems.

A widely used neuron model is the *perceptron*. Mathematically, a perceptron is a function $p:\mathbb{R}^m \to \mathbb{R}^1$, it takes m inputs and outputs a single number. The perceptron model further decomposes p into two components, $p = s \circ c$, a *combiner* (c) that transforms the m-dimensional input vector $\boldsymbol{x} = (x_1 .. x_m)$ into a single number and a *saturator* (s) that saturates the output from the combiner.

Figure 3.1 – The Perceptron

In the simplest case, the combiner is a linear combination of inputs:

$$\begin{aligned} c(\boldsymbol{x};\boldsymbol{w},b) \equiv & \quad w_1 x_1 + w_2 x_2 + \ldots + w_m x_m + b \\ = & \quad \Sigma w_i x_i + b \\ = & \quad \boldsymbol{w}\,\boldsymbol{x} + b \end{aligned}$$ - Equation (3.1)

Where **w** are the "weights" of the combiner and b is a "bias". These numbers completely define the linear combiner. Equation 3.1 is just the inner product between the vectors **w** and **x** (with a bias added to it), and this is what Caffe calls this component (or "layer", in Caffe parlance).

The saturator can either be the "sigmoid" function or the hyperbolic tangent. We'll use the hyperbolic tangent (tanh) in our work:

$$s(z) = \tanh(z)$$ - Equation (3.2)

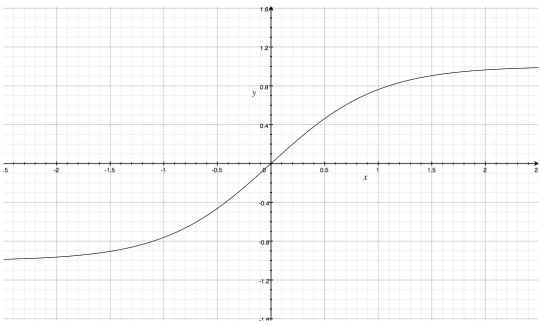

Figure 3.2 – Hyperbolic Tangent

A saturator needs to satisfy a few criteria:

1. It is a fixed function. So, all learning/training happens only in the combiner, by selecting appropriate weights **w**.
2. It has a "linear response" region. In this region, the saturator outputs a signal roughly proportional to its input. (This criterion is important for training.)
3. Outside the linear response region, the signal is saturated, reaching maximum/minimum asymptotically.
4. The saturator is increasing in its input.

Clearly, from Figure 3.2, the hyperbolic tangent fits the bill – it is a monotonically increasing function, saturating at -1 and +1, with a linear

response region roughly within [-0.5 , 0.5]. Many other functions can be used too, the sigmoid function being a common choice. So, in our case, the perceptron can be neatly summarized as:

$$p(\boldsymbol{x}) = \tanh(\boldsymbol{w}\boldsymbol{x} + b) \quad \text{- Equation (3.3)}$$

This equation tells us a few things:

- A single perceptron assigns to *most* points in an m-dimensional input space either -1 or +1, since most points **x** will cause it to saturate.

- Since the combiner is linear in **x**, this will divide the m-dimensional input space with a single hyperplane, parameterized by **w** and *b*. For example, for a 1-dimensional input space (eg, only temperature, *T*), the combiner is $c(T) = wT + b$, which is a straight line. For a 2-dimensional input space (eg, temperature T and humidity *r*), $c(T, r) = w_1 T + w_2 r + b$, which is a plane. Points well "above" (ie $c(T, r) \gg 1$) the plane will be assigned +1 while points well below ($c(T, r) \ll -1$) are assigned -1.

- So, by themselves, single perceptrons are very limited in what functions they can represent. A single perceptron splits the input space in two with a single hyperplane, coloring one side +1 and the other -1. Close to the hyperplane, the color is "fuzzy" (between -1 and +1), since that is the linear response region of the saturator.

3.2 Why do they work?

While individually limited, it is possible to prove that you can add together the outputs of many perceptrons to approximate *any* **continuous** function.

Let's get an intuitive feel for this in 1-dimension input space. Consider the following:

- Any continuous 1-dimensional function $f(x)$ can be approximated to arbitrary accuracy using a sum of localized pulses (eg, square pulses or single peaks). This is very much like using a "histogram" to approximate a function in basic Riemann integration.

- A 1-dimensional localized pulse can be constructed as the sum of two perceptron outputs. We'll call these the "right" and "left" perceptrons. For example, you can construct a pulse with $\tanh(x+0) + \tanh(-x+1)$. Note that the biases 0 and 1 set the offset of the peak along the x-axis,

in this case at x=0.5

- To obtain higher peaks, simply multiply these pulses by a weight. Figure 3.3 plots out a higher peak using this trick (weight = 5).

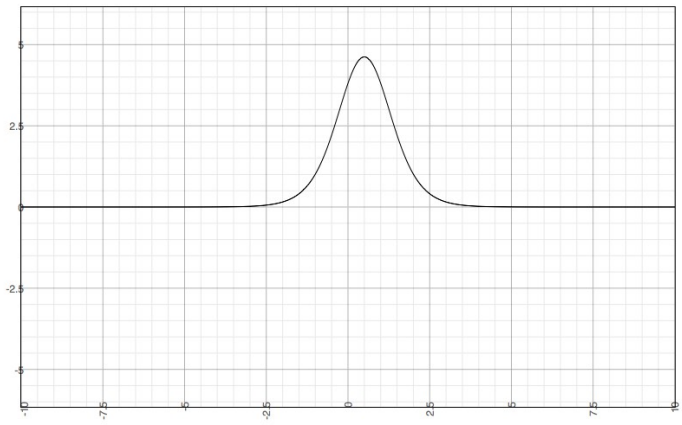

Figure 3.3 - $5\tanh(x)+5\tanh(-x+1)$

Stepping back, we see that this architecture in Figure 3.4 produces a localized pulse.

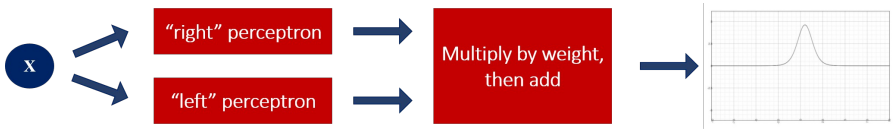

Figure 3.4 – Creating a localized pulse from two perceptrons

There are a couple of important things to note:

- The "right" and "left" perceptrons work as a single unit to produce a pulse waveform.
- The pulse waveform is nonzero only for a small, connected region on the x-axis, $[x_1, x_2]$. In Figure 3.3, this is roughly [-2.5 , 3.0].

This is a simple example of a *feature detector*, that is, a collection of neurons that activate together (give a non-zero output) only if the inputs fall within a small range. Feature detectors are important for generalization, which we will discuss in-depth in future lectures.

Let's call this right/left perceptron unit a “pulse generator”. The pulse generator really only needs to know the offset of the pulse; the weights of the individual perceptrons are fixed at +1 and -1 respectively. So, to construct an arbitrary function, we just need to scale pulses then add them all up:

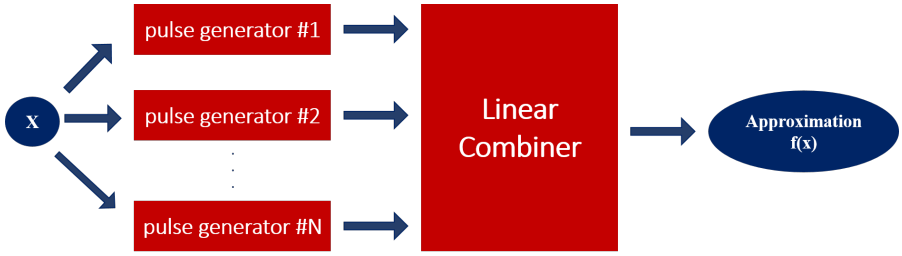

Figure 3.5 – A collection of pulse generators with a linear combiner can approximate any continuous 1-D function.

This architecture in Figure 3.5 is equivalent to a single layer of perceptrons, all connected to the same input. The output of these perceptrons are sent to a single linear combiner to calculate the final output.

Quiz 3A

1) Construct a 1-dimensional pulse generator (ie, define all weights and biases of each perceptron) where the peak of the pulse is at x = 3.
2) Explain why the saturation property of perceptrons is crucial to forming a pulse.
3) * Extend your intuition for the functional approximation property of perceptrons, from 1-dimensional input to 2-dimensions. Can you extend it to N-dimensions?
4) The approximation is only valid for continuous functions. Can you give an example of a discontinuous function that can't be approximated this way using a single layer of perceptrons?
5) Rama wants to construct a sine wave generator, $y(t)=\sin(t)$ and $t\in[-\infty,+\infty]$ using a finite number of perceptrons and a linear combiner. Can this be done? Explain your reasoning.

3.3 The Learning Problem

Is there a way to construct functional approximations “automatically” instead of “by hand” as in the preceding section? This is the essence of the *Learning*

Problem, which is the main subject of this course.

To solve the learning problem you are given a sample of input/output pairs to learn from. This is called the *training set*. It is quite easy to get a model to approximate the training set – just memorize the pairs! What is desired however is the ability to *generalise* outside the training set.

To put it more concretely, the fundamental problem we tackle is predicting the time behaviour of an unknown function $f(\boldsymbol{x}(t))$ given a sample D of training pairs $D=\{f(\boldsymbol{x}_k), \boldsymbol{x}_k\}$. Caffe calls $f(\boldsymbol{x}_k)$ the "labels" and $\boldsymbol{x}_k$ the "data".

- Our goal is to construct a NN model using D so that the model doesn't just correctly emulate $f(\boldsymbol{x}_k)$ for $\boldsymbol{x}_k \in D$, but also outside D. A NN able to do this "successfully" is said to be capable of generalization.

- We measure "success" using a *loss function* $L:\mathbb{R}\times\mathbb{R}\rightarrow\mathbb{R}$ against a set of testing pairs E. The lower the loss over E the better our model is able to generalize.

- In order to measure successful learning of the training set D, we use a *risk functional*, which is the expected losses over the training set D.

So, using just D, and R, we somehow want to amend the model's parameters so that it minimizes the loss on E. This "somehow" is the job of the *learning algorithm*.

Before we proceed we need to look at loss and risk functions a little closer since they play a crucial role in time-series forecasting.

3.4 The Loss Function

A simple loss function is the *Euclidean loss*:

$$L(f_k, \hat{f}_k) \equiv (f_k - \hat{f}_k)^2$$

where $\{f_k\}$ are labels and $\{\hat{f}_k\}$ are the model's corresponding predictions. Some important characteristics of a loss function L are that:

- L has a minimum when an optimal solution a is reached, ie, $L(a,x)$ is a (global) minimum at $x=a$. This is to facilitate the use of minimum-finding algorithms to locate optimal solutions. Preferably, we want this minimum to be the only one.

- L should be continuous almost everywhere. This is so that we can use local algorithms (like gradient descent) to iteratively improve on an approximate solution and incrementally move towards the optimal solution. For the Backpropagation algorithm, we need a stronger condition, namely that the loss function is differentiable almost everywhere.

Quiz 3B:

1. Propose an alternative loss function.
2. Explain clearly why the loss function must have a global minimum.
3. * Why should this minimum be the only one?
4. * Why should the value of the minima be at $x = a$?
5. * What happens if the loss function is discontinuous near an optimal solution?

3.5 The Risk Functional

The expected value of the loss function over the training set D is known as the *risk functional* or just *risk*. The risk functional R is approximated as a simple average over the losses of the training set. For the Euclidean loss:

$$\begin{aligned} R(\{f_1 .. f_N\}, \{\hat{f}_1 ... \hat{f}_N\}) \equiv & \quad E_D[\![L(f_k, \hat{f}_k)]\!] \\ \approx & \quad \frac{1}{N} \sum_D L(f_k, \hat{f}_k) \\ = & \quad \frac{1}{N} \sum_{k=1}^{N} (f_k - \hat{f}_k)^2 \end{aligned}$$

$E_D[\![..]\!]$ is the expectation value over the set D. Like the Loss function, the risk functional should also have two same desirable properties:

- A single and global minimum around the optimal solution $\boldsymbol{a} = \{f_1 ... f_N\}$
- Continuity of the risk functional almost everywhere.

3.6 Risk Minimization

Most learning algorithms work by seeking model parameters that minimize the risk functional over the training set D.

The basic idea is that by minimizing risk (by changing the model's parameters), we are therefore minimizing the average loss. This should yield a good solution.

For example, in the case of a single perceptron, the “model parameters” are just the perceptron's weights **w**. Risk minimization then involves somehow changing these weights in order to minimize risk. A learning algorithm like Backpropagation works by performing risk minimization.

The idea is simple but subtle. You must be very careful of two things:

1. That the final goal of risk minimization does indeed produce an optimally *trained* network. This can be trivially checked if the loss & risk have a single global minimum.

2. That the optimal solution achieved by risk minimization conforms to your mental picture of a “good” solution. This is very hard to do and for time-series predictions, it can be a cause of failure (see Quiz 3C.4 below). This is because the risk functional doesn't just evaluate the goodness of a model. In local algorithms like Backpropagation, it is also acts like the “eyes” of the algorithm, enabling it to find a good solution to the Learning Problem.

A bad loss function or risk functional can doom your Backpropagation learning to failure. This is especially so for time-series forecasting. We'll revisit this in depth in Lecture 6 tomorrow.

Quiz 3C:

1. Explain why the risk functional should have the same desirable properties as the loss function.

2. * Lim wants to use the *maximum* loss over *D* instead of the average loss in his new Risk Functional definition: $R \equiv max_D[\![L(f_k, \hat{f}_k)]\!]$. Do you think this is a good idea? Explain your answer clearly.

3. ** Ali wants to tackle risk minimization by splitting the training set *D* into two sets mutually exclusive sets, $D = D_1 \cup D_2$. His idea is to perform risk minimization in two passes: first over D_1 , use that solution as a starting point to minimize risk over D_2 . Do you think this is a good idea? Explain your answer clearly.

4. * Cathy wants to predict rare events. What problems should she anticipate using the risk functional as we've defined it? Should she focus on an alternative risk functional or change her loss function?

3.7 Feedforward Networks

A *path* is an ordered set of neurons $\{s_0, s_1 \ldots s_m\}$, where the output of s_k is the input into s_{k+1}. A path is *recurrent* if additionally, the output of s_m is the input into s_0. In other words a recurrent path represents a cycle of neuronal connections.

Architectures containing a recurrent path are called *recurrent networks*, while those without are called *feedforward networks*. This distinction is important for the Backpropagation learning algorithm, which is only stable for feedforward networks. In fact, in practice, it performs best when the longest path is just a few neurons in length.

3.8 The Backpropagation (BP) Algorithm

Most learning algorithms work by minimizing risk. In the case of the Backpropagation (BP) algorithm, this is done using gradient descent. The BP algorithm is useful because it can be computed efficiently and because it can be easily parallelized. Figure 3.6 shows a generic network:

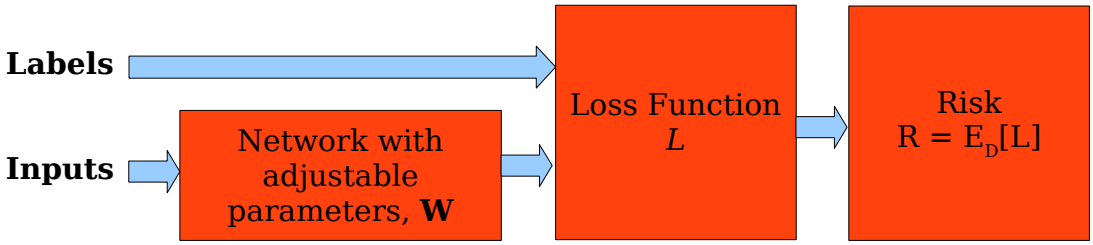

Figure 3.6 – A generic network with adjustable parameters.

Backpropagation starts with the observation that if risk is minimized for a set of parameters W^*, then by definition,

$$\left.\frac{\partial R}{\partial W}\right|_{W=W^*} = 0$$

So, to find the minimum parameters W^* starting from a trial W_{old}, one option is to use *gradient descent*:

$$W_{new} = W_{old} - \eta \left.\frac{\partial R}{\partial W}\right|_{W_{old}} \quad \text{– Equation 3.1}$$

Where η is a "small" number known as the *learning rate*. Under certain

conditions, iterating this algorithm and ensuring $\eta \to 0$ will give a solution W^*. More sophisticated gradient descent algorithms (like the ones available in Caffe's Solver) work in roughly the same way.

The key here is to calculate the term $\frac{\partial R}{\partial W}$ for our network. Then the network's parameters are updated using Equation 3.1. You only need to tell the solver when to stop. Now,

$$\frac{\partial R}{\partial W} = \frac{\partial E_D[\![L]\!]}{\partial W} = E_D[\![\frac{\partial L}{\partial W}]\!] \quad \text{– Equation 3.2}$$

The term $\frac{\partial L}{\partial W}$ is really shorthand for $\frac{\partial L}{\partial W_j^\alpha}$, where W_j^α is the *j*-th parameter for a neuron s_α. Consider a path $\{s_\alpha, s_{\alpha+1} \dots s_N\}$ that starts with s_α, and ends with s_N, a neuron that outputs the prediction. L can therefore be written as:

$$L = L(X_N) = L(X_N(W^N; X_{N-1}))$$

Where X_N is the output of neuron s_N and W_N its parameters. Using the chain rule once, we have:

$$\frac{\partial L}{\partial W_j^\alpha} = \frac{\partial L(X_N)}{\partial W_j^\alpha} = \frac{\partial L}{\partial X_N} \frac{\partial X_N}{\partial W_j^\alpha} \quad \text{– Equation 3.3}$$

The term $\epsilon = \frac{\partial L}{\partial X_N}$ depends only on the form of the loss function. Its value should vary according to the model's performance over the entire training set. Also, since W^α and W^N are independent variables, we have:

$$\frac{\partial X_N}{\partial W_j^\alpha} = \frac{\partial X_N(W^N; X_{N-1})}{\partial W_j^\alpha} = \frac{\partial X_N}{\partial X_{N-1}} \frac{\partial X_{N-1}}{\partial W_j^\alpha}$$

The term $\delta_N = \frac{\partial X_N}{\partial X_{N-1}}$ measures how much the output of neuron s_{N-1} affects the output of neuron s_N. We can therefore rewrite Equation 3.3 as:

$$\frac{\partial L}{\partial W_j^\alpha} = \epsilon \frac{\partial X_N}{\partial W_j^\alpha} = \epsilon \delta_N \frac{\partial X_{N-1}}{\partial W_j^\alpha}$$

From this, it shouldn't be too hard to convince yourself that:

$$\frac{\partial L}{\partial W_j^\alpha} = \epsilon \delta_N \delta_{N-1} \ldots \delta_{\alpha+1} \frac{\partial X_\alpha}{\partial W_j^\alpha}$$ – Equation 3.3

We call the term $\epsilon_{\alpha+1} = \epsilon \delta_N \delta_{N-1} \ldots \delta_{\alpha+1}$ the error received by neuron s_α from the path $\{s_{\alpha+1} \ldots s_{N-1}, s_N\}$. You can see from its form that the original error ϵ is propagated backwards from s_N into s_α. From this, we can rewrite Equation 3.3 as:

$$\frac{\partial L}{\partial W_j^\alpha} = \epsilon_{\alpha+1} \frac{\partial X_\alpha}{\partial W_j^\alpha}$$

$$\epsilon_{\alpha+1} = \epsilon \delta_N \delta_{N-1} \ldots \delta_{\alpha+1}$$

$$\delta_k = \frac{\partial X_k}{\partial X_{k-1}}$$

$$\epsilon = \frac{\partial L}{\partial X_N}$$

These four equations define the Backpropagation (BP) algorithm. Figure 3.7 below depicts this flow of error terms:

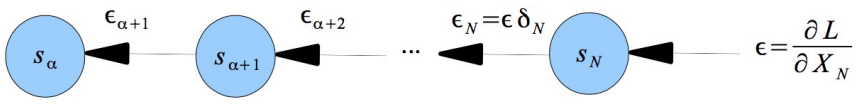

Figure 3.7 – Error flow in the BP algorithm.

In the event that a neuron is the root node of more than one path, then the chain rule means that you need to sum the errors from each path:

$$\epsilon_{\alpha+1} = \sum_{1..p} \epsilon_{\alpha+1}^{(p)}$$

Quiz 3D.1 Show this is true.

3.9 Batch Learning

In *batch learning* or *batch updating* , the weights are only updated once the risk has been calculated for the entire training batch.

Quiz 3C.3 discusses an alternative form of update called “mini-batch” learning. An extreme case is “stochastic” learning” where the weights are updated after receiving each input. Batch learning generally provides the best convergence, and we recommend you start with this. Stochastic learning is generally very “noisy” with sharp peaks in the training loss over time. However, it is said this provides better escape from local minima of gradient descent and much faster convergence.

To recover the BP equations involving Risk, we simply have to take the expected value of these equations over the training set *D*:

$$\begin{aligned} \frac{\partial R}{\partial W_j^\alpha} &= E_D[\![\epsilon_{\alpha+1} \frac{\partial X_\alpha}{\partial W_j^\alpha}]\!] \\ \epsilon_{\alpha+1} &= \epsilon \delta_N \delta_{N-1} \ldots \delta_{\alpha+1} \\ \delta_k &= \frac{\partial X_k}{\partial X_{k-1}} \\ \epsilon &= \frac{\partial L}{\partial X_N} \end{aligned}$$

Equations 3.4 – The BP Equations for Batch Learning

3.10 Caffe's Forward & Backward Passes

During each training phase iteration, your Caffe model makes 2 *passes*, one after the other.

1. A *forward pass* is used to generate a set of predictions based on the training input (see Figure 2.3 from Lecture 2). The predictions are used to compare against the labels using your model's loss layer(s).

2. During the *backward pass* Caffe's solver computes the error $\epsilon = \frac{\partial L}{\partial X_N}$ from the loss blob(s) and passes it backward as an error, starting from the end nodes of your network, layer by layer down your network. Each time an error ϵ is received by a layer, it uses it to correct its weights (if necessary) using the BP algorithm (Equations 3.4), and then passes down that error ϵ multiplied by its own error term δ . That is, it passes down $\epsilon' = \epsilon \delta$. In the event that layer receives multiple error

signals, $\{\epsilon_1 \ldots \epsilon_n\}$, these are added up by the layer, ie, $\epsilon = \sum_{1..n} \epsilon_k$ prior to performing the BP weight update.

As a simple example, consider a network consisting of just one perceptron (Figure 3.8). This would consist of 2 Caffe layers, an `InnerProduct` layer (with 1 output) and a `TanH` layer. Let's use a `EuclideanLoss` layer as our loss layer (in fact, the output of any layer can be used as a “loss” layer).

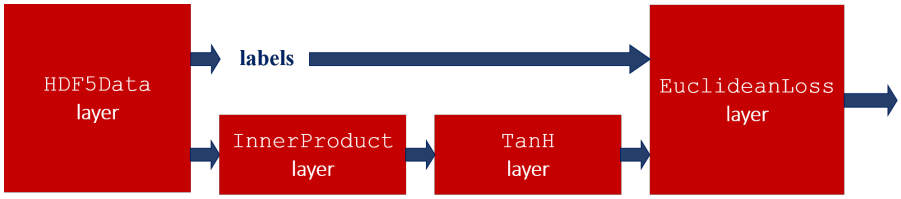

Figure 3.8 – A really simple NN model

Here's a detailed breakdown of what happens during one training phase iteration (1 forward and 1 backward pass):

1. ***Forward Pass***: The input blob – say (720 3) is transformed by the network to predictions (720 1). This is combined with labels (720 1) to form the network's losses (720 1).

2. ***Backward Pass***: The error ϵ is calculated from the loss blob(s). This is passed first backward to the network's `EuclideanLoss` layer, but this does nothing but send the error to downstream layers unchanged, since this layer is stateless.

3. The downstream layers are are a `HDF5Data` layer (that read in the labels), that does not do anything with the error, and the `TanH` layer from the perceptron.

4. This `TanH` layer is stateless (and therefore, has nothing to update), but it computes it's error term and passes $\epsilon\delta_{\tanh}$ backward to the the `InnerProduct` layer.

5. This `InnerProduct` layer does have weights and amends them using (i) the current learning rate (ii) the received error term $\epsilon\delta_{\tanh}$.

6. The `InnerProduct` layer computes it's error term δ_{IP} and passes backward the total error $\epsilon\delta_{\tanh}\delta_{IP}$ to the layer below it, a `HDF5Data` layer

that read in the inputs data. This layer does nothing with the errors, simply discarding them.

Note that in batch learning, weight updates are made only after the entire training batch has been processed. The weight update is proportional to:

$$\frac{\partial R}{\partial W_j^\alpha}$$

In stochastic learning, the update is made after every training datum is received. The weight update is proportional to:

$$\frac{\partial L(t)}{\partial W_j^\alpha}$$

where *t* is the *t*-th training datum.

Caffe allows you to select either batch or stochastic learning or mini-batches (where the weight updates are done after subsets of the training set is seen) using the `batch_size` parameter. For batch learning, set the `batch_size` to be the same as the size of the training set. For stochastic learning, set the `batch_size` to be equal to 1.

3.11 Problems with BP

In practice, the BP algorithm is good for shallow networks (1 – 3 layers deep) but has shortcomings for anything much deeper. The errors become small (and therefore prone to numerical noise) or large (in which case the training will be poor.) The reason is simple. Along a path $\{s_0, s_1 \ldots s_m\}$, the total error term becomes:

$$\epsilon_0 = \epsilon \prod_{1..m} \delta_k \quad \text{- Equation (3.5)}$$

If it happens that $\delta_k > 1$ and *m* is large (as it happens in deep layers or with recurrent connections), the total error ϵ_0 will blow up. Conversely, if $\delta_k < 1$, the it will vanish.

In practice, either due to large weights (causing error blowup) or saturation (causing vanishing error), it doesn't take much for either situation to happen. (**Quiz 3D.2** Show how saturation in “upstream” neurons causes vanishing errors downstream)

The problem is especially problematic for recurrent networks (**Quiz 3D.3** Can you see why?) In Lecture 10 we will briefly touch on LSTMs, a special non perceptron-based architecture that has recurrent connections but can be trained using BP.

Also in Lecture 7 we will cover a technique called *greedy layer-wise training* which can mitigate these problems for feedforward networks.

3.12 A Real-world Example: Dengue in Singapore

In the lectures and Labs that follow, we will be using Dengue prediction as an example of a useful, real-world problem.

In this problem, we want to predict the number of dengue cases in Singapore 16 weeks from the present ($f(\boldsymbol{x})$) , given the temperature (T), rainfall (r) and dengue cases (d) in the present. So, $\boldsymbol{x}(t)=(T(t),r(t),d(t))$. In this case, $f(\boldsymbol{x}(t))\equiv d(t+16)$.

To do this, we need a training and test sets D and E, and a loss function L. Typically, we use 70% - 90% of our data for training (D) and the last 30% - 10% for testing (E).

This is an open-ended problem – we will not be offering you any answers (although we have largely solved this problem). The Dengue Prediction problem has many important characteristics of time-series problems you might encounter, and we hope, makes for excellent learning.

Quiz 3E:

1. What do you think motivated us to select a 16-week time horizon? How can the way we made our choice inform your time-horizon selections?

2. Do you think the training inputs are sufficient to predict dengue? Explain your thinking.

3. * Calculate the BP equations for the neural network in Figure 3.8

4. John wants to include a new layer into Caffe, called InputNormalization. This layer scales does subtracts the mean and scales it by the variance: $y_k=(\frac{x_k-\mu_k}{\sigma_k})$, with y_k being the outputs, x_k the inputs, μ_k the

input means and σ_k the input variances. Calculate the BP error contribution δ for this layer. What are the adjustable parameters for this layer?

Lab 3: Predicting the mean temperature 1, 4, 8 weeks ahead.

In this lab, you will:

1. Construct & train a simple feedforward network to predict mean temperature 1, 4 and 8 weeks ahead of the present.
2. Use Autocaffe to “expand” your networks.
3. Visualize your results using Autocaffe's reporting facility.

Instructions

We build on the NN set up used for Lab 2A, this time applied on real-world temperature data.

Input signal

From the plot of the weekly mean temperature (fig L3.1), you can see that, unlike the smooth sine wave, there are short term variations on top of the periodic pattern, as well as a slight general upward trend. Due to the greater short term variation, for the NN to be able to learn the longer term pattern, it needs to have a larger input widow to look at as compared to a clean signal. In this lab we use a window size of 20.

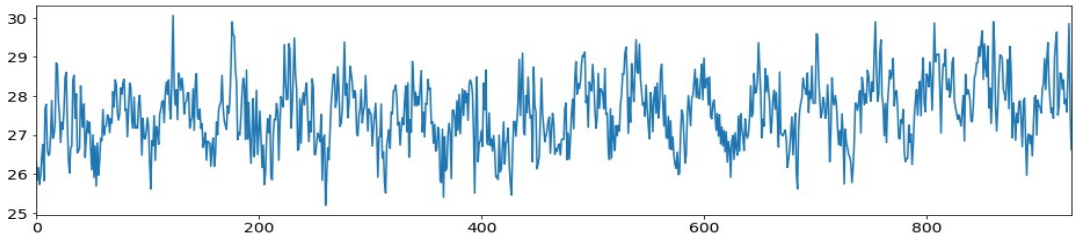

Fig L3.1: Mean temperature plot

The mean temperature ranges from about 25 to 30, which is large compared to the tanh saturator that we use. This makes it harder for the NN to learn as it first has to learn to increase its output to such high levels. The NN may also end up operating in the saturated region of the tanh curve (a region where output becomes insensitive to input) in an attempt to produce larger outputs. Similarly, NNs also learn better if the inputs are not excessively large. The way to avoid these problem is through input normalisation, a topic covered in Lecture 5. For this lab, we will simply subtract the mean from each value in the dataset.

Dataset

The dataset has been prepared for you (located at `~\caffe\data\mean-`

temp\) , but you can try to recreate it yourself from the aggregated weekly dataset that you made in the take-home Lab 2B.

Create a new folder named lab3 in the experiments folder (~\caffe\experiments). **Do not put any space or underscores in the folder name.** Create three new files named train.txt, test.txt and val.txt files in that folder and copy the paths given below for each file.

- train.txt: \home\terra\caffe\data\mean-temp\train.h5
- test.txt: \home\terra\caffe\data\mean-temp\test.h5
- val.txt: Both train.h5 and test.h5

The dataset **data** stored in train.h5 and test.h5 has dimension of (723 20) and (181 20) respectively. The 723 and 181 values of the first axis are the number of examples in the train and test set and are determined by a 80/20 split between the train and test sets. The value 20 in the second axis means that for each example, 20 values are fed to the NN. This corresponds to the size of the input window.

The label dataset has dimension of (723 3) and (181 3) in train.h5 and test.h5. The first axis has to be identical to those in the data dataset, while the second axis has 3 numbers corresponding to the T+1, T+4 and T+8 values of the temperature. In an experiment, you will only use one of the three label values in the second axis, so you need to slice off the other two values using a Caffe layer in the prototext NN structure definition file.

Autocaffe expander

Trying out many different variations of a NN structure can be a bit tedious. Suppose you want to vary the number of nodes in the inner product layer. One way to do that is by manually making multiple experiment folders and editing the prototext files with different inner product layer sizes. Instead of doing that, the expander function of autocaffe allows you to specify a variable representing the layer sizes that you want to try out. Then with a single prep command, the experiment folder is expanded with the layer sizes filled in.

To do this, you need to create an additional text file called config.txt in the experiment folder (~\caffe\experiments\lab3). In it, you enter all the variables you want and their respective ranges. For this lab, you will want to try different inner product layer sizes. Put this line in your config file:

```
SIZE : #{ 8 32 128 512 4 list as SIZE } ${ $ SIZE . }
```

This consists of two commands to autocaffe.

```
#{ 8 32 128 512 4 list as SIZE }
```

This command tells autocaffe that the `SIZE` variable contains 4 values: 8, 32, 128 and 512.

```
${ $ SIZE . }
```

This command will be read by autocaffe and replaced with the value of SIZE assigned for the particular experiment.

To do the expansion, run the usual prep command (after you have prepared all the files needed for the experiment to run):

```
ac prep lab3
```

In this example, 4 folders will be created in the results folder, each with one of the four `SIZE` values.

NN structure

Use the layers shown below for your `train.prototxt` file. We introduce a few more Caffe layer types. You can find information about each layer (its uses and parameters) from the Caffe layer catalogue.

Input layers: These layers and are the same as in lab 2A, aside from the `batch_size` parameter setting of 723 (you may try other values as well). This parameter sets how many input examples are read in an iteration. For a batch size of 100, the output dimension of this layer will be (100 20) – ie in the first iteration, the first 100 examples of 20 values each are output, on the next iteration the 100th to 200th examples are output, etc. This is called mini-batching and is discussed in this lecture. We set the batch size of the test phase layer to the size of the test dataset (181) so that the test phase can be completed in one iteration.

```
layer {
	name: "input"
	type: "HDF5Data"
	hdf5_data_param {
		batch_size: 723
		source: "train.txt"
	}
	top: "data"
	top: "label"
	include: {
		phase: TRAIN
	}
```

```
}

layer {
	name: "input"
	type: "HDF5Data"
	hdf5_data_param {
		batch_size: 181
		source: "test.txt"
	}
	top: "data"
	top: "label"
	include: {
		phase: TEST
	}
}
```

Normalisation: After the input layer is the `Power` layer. The power layer does element-wise operations on its input to give output $y=(shift+scale*x)^{power}$ with `shift`, `scale` and `power` being adjustable parameters. These parameters have default values of 1 for `scale` and `power`, and 0 for `shift` if they are not defined in the prototext file. We use this layer for input normalisation by subtracting the mean value (27.5) from the `data` dataset:

```
layer {
	name: "subtract"
	type: "Power"
	bottom: "data"
	top: "subbed_data"
	power_param {
		shift: -27.5
	}
}
```

InnerProduct layer: This layer performs an inner product (plus a bias) between its internal learnable weight matrix and its input.

The `weight_filler` and `bias_filler` parameters specify how the weights in the matrix and the biases are initialised before training begins. In this case, the weights are filled using the Xavier method, which randomly fills the weights within certain bounds. This randomness means that the same NN structure starts at different points in its configuration space and may end up with different results when trained multiple times.

The `axis` specifies the first axis that will be lumped together as an input vector for the inner product operation. Since the axis is set to 1 and the input has dimension (100 20), the layer performs an inner product between the size

20 vector of each example and its weight matrix. If the axis was set to 0, then all 100*20 values in the input blob will be multiplied with the matrix in a single inner product operation.

The `num_output` parameter gives the size of the vector after the inner product operation. Continuing the example above, if `num_output` is 5, then the layer output will have dimension (100 5). If the `axis` parameter is 0, then the output dimension will be (5).

Instead of typing the desire `num_output` value, we use the expander command `${ $ SIZE . }`. As mentioned earlier, this will be substituted with the assigned `SIZE` value in the prototext file itself when the experiment is `prep`-ed over to the results folder.

```
layer {
	name: "inner-product"
	bottom: "subbed_data"
	top: "ip"
	type: "InnerProduct"
	inner_product_param {
		num_output: ${ $ SIZE . }
		axis: 1
		weight_filler {
			type: "xavier"
		}
		bias_filler {
			type: "constant"
			value: 0
		}
	}
}
```

TanH layer: This layer applies the TanH function element-wise on its input blob called `ip` and produces an output blob `t`.

```
layer {
	name: "tanh"
	type: "TanH"
	bottom: "ip"
	top: "t"
}
```

Output layer: The next layer convert the SIZE value input of dimension (100 5) into a single prediction value of dimension (100 1). This layer does not need an TanH layer applied to its output.

```
layer {
	name: "output"
	bottom: "t"
	top: "raw_prediction"
	type: "InnerProduct"
	inner_product_param {
		num_output: 1
		axis: 1
		weight_filler {
			type: "xavier"
		}
		bias_filler {
			type: "constant"
			value: 0
		}
	}
}
```

Power layer: The layer relieves the NN from having to produce large values by back adding the mean value that was substituted from the input at the start to give the final prediction value.

```
layer {
	name: "add"
	type: "Power"
	bottom: "raw_prediction"
	top: "prediction"
	power_param {
		shift: 27.5
	}
}
```

Slice layer: The label dataset as described above contains the T+1, T+4 and T+8 values in the second axis (`axis=1`). To use one of these to use and discard the others, we have to slice the `label` blob into three along the axis 1. In this case, the blob is sliced at two points to produce three outputs, each of dimension (100 1).

```
layer {
	name: "slice"
	type: "Slice"
	bottom: "label"
	top: "label_1"
	top: "label_4"
	top: "label_8"
	slice_param {
		axis: 1
```

```
                slice_point: 1
                slice_point: 2
        }
}
```

Silence layer: This layer "silences" the T+1 and T+8 label blobs that is not used. Blobs which are not silenced or consumed by another layer will be printed in the log (`train.log` or `test.log`) as an output of the NN.

```
layer {
        name: "Silence"
        type: "Silence"
        bottom: "label_1"
        bottom: "label_4"
}
```

EuclideanLoss layer: This layer calculates the loss between its two inputs. Note that the output blob `loss` is not consumed by any other layer and the loss is thus printed in the `train.log` file after every n iterations, where n is the `display` setting in `solver.prototxt`.

```
layer {
        name: "loss"
        bottom: "prediction"
        bottom: "label_8"
        top: "loss"
        type: "EuclideanLoss"
}
```

NN structure – `test.prototxt`
For the test network, use the same layer structure and make the following modifications

- Remove the train phase input layer: this is the `HDF5Data` layer with the `include: { phase: TRAIN } }` parameter.
- Input layer: change the `batch_size` of the (test phase) input layer to 904 (which is the size of the combined train and test datasets) and the `source` to `val.txt`.
- Remove the Euclidean loss layer. This is to allow the `prediction` and `label` blobs to pass on into the `test.log` file where it is extracted to produce the `predictions.csv` file

Note: The names of the inner product layers **must be the same** in both

`train.prototxt` and `test.prototxt` networks. This is to enable the weights and biases of the inner product layers that were learned during the training process to be retrieved and used for the test network. If the layers are named differently, Caffe will use random initialised weights and biases, and you will not be testing your trained model. The other layers (eg Slice or Power layers) don't have learned weights and hence do not need to have the same name.

Solver

Use the same settings as in Lab 2A, but set `test_iter` to 1, since the batch size in the test phase is equal to the size of the test dataset. You may want to change the base learning rate (`base_lr`), and the `max_iter` (remember to set `snapshot` to whatever your `max_iter` is, so that only the fully-trained model is saved), or try a range of values using the expander and see their effects.

report.tex_template

Copy the following latex code for `report.tex_template`

```
\section{Experiment ${ $ name . }}

Size: ${ $ SIZE . } \\
Train loss: $LOSS_TRAIN \\
Test Loss: $LOSS

\begin{figure}[H]
\centering
\includegraphics[width=\textwidth]
{${ $ pwd (.) }figs/prediction.png}
\caption{Prediction}
\end{figure}

\begin{figure}[H]
\centering
\includegraphics[width=\textwidth]
{${ $ pwd (.) }figs/test_prediction.png}
\caption{Prediction (test set)}
\end{figure}

\newpage
\begin{figure}[H]
\centering
\includegraphics[width=\textwidth]
{${ $ pwd (.) }figs/loss-training.png}
\caption{Training Loss}
\end{figure}
\newpage
```

A few expander commands have been used here

- `${ $ name . }` : This prints the experiment folder name
- `${ $ pwd (.) }` : This prints the path to the experiment folder

These commands are required since the experiment will be expanded into several different folders.

The `$LOSS_TRAIN` and `$LOSS` words will be substituted with the loss over the train and test datasets respectively by the `post` command used later.

metrics
Copy the metrics file provided in `~/caffe/datathon-files/lab3/` into your experiment folder. This file is used for the `post` command to print the loss values into the `report.tex` file.

Job scheduling
Prep the experiment as usual:

```
ac prep lab3
```

Add all expanded experiments to the jobs queue at once with

```
ac add lab3/*
```

Wait for all jobs to complete. Check by doing `ac q` and `ac list` and making sure that there are no entries for both. Then run the test network with

```
ac test lab3/*
```

Use the post command to calculate the loss and substitute them into the report

```
ac post lab3/*/metrics
```

Finally, generate the pdf report using

```
ac report lab3/*
```

Experiment with different configurations like inner product layer size, try to increase the depth of the network by adding more inner product and tanh layers. If the training or test loss decreases too quickly (within one iteration) or is still decreasing at the end, try changing the learning rate by a factor of

10. Look at the loss and predictions plot and see if you can make sense of your results.

Quiz 3F

1. How did the test cases perform for larger time horizons?
2. How much of the temperature dataset (input window size) did you use for training/testing? What drove your choice? Explain your thinking.
3. How good was your final training loss? Final test loss? Was there any correlation between the two for the various configurations? Explain your results.
4. Do the plotted graphs for the test phase match your expectations?
5. Was there any improvement using deeper networks? As best as your can, explain your results as you increased your layers, using the lecture materials to date.
6. Would the optimum learning rate change with network size? Explain your reasoning and try to set up an experiment to demonstrate your hypothesis, if applicable.